EPISODE 872

[INTRODUCTION]

[00:00:00] JM: Facebook was built using PHP, a programming language that was widely used in the late 90s and early 2000s. PHP allows developers to get web applications built quickly and easily, although PHP has a reputation for being difficult to scale.

In the early days of Facebook, the company was scaling rapidly on every dimension. New users were piling into Facebook. Existing users were increasing their interactions and developing new patterns of usage. The Facebook application was rolling out new features quickly, adding them into the Facebook PHP codebase.

A common pattern for scaling a large software application is to use a microservices architecture breaking up the monolithic application into small services which can scale independently. For many applications, this pattern works well. But for some applications, microservices makes less sense. Microsoft Excel might be one example.

In Excel, a user is making updates to a complex data model using formulas, functions and other in-app tools that need to be fast, performant and integrated. The user needs to have a sense that the Excel data model will update quickly in response to changes. A software team working on a spreadsheet product, such as Excel, might prefer to keep all the application logic in a monolithic application.

A monolith can centralize logic and make it easier to reason about. A monolith can reduce the number of network cops, cutting down on distributed systems problems. Testing and deploying a monolithic application can be less complex than doing so in a distributed microservices system.

Facebook chose to scale its PHP monolith rather than breaking it up into distributed microservices. Scaling PHP allowed Facebook to continue moving fast without going through a painful refactoring that would have slowed down the entire company. The first effort to scale PHP involved transpiling the entire PHP application into C++.

This C++ version of Facebook ran faster and with a lower memory footprint. But C++ required ahead of time compilation. The PHP codebase had to be converted to C++ in one synchronous step.

The Hip Hop Virtual Machine (HHVM) is a just-in-time compiler that serves as an execution engine for PHP as well as Hack, a language that Facebook created as a dialect of PHP.

HHVM allows for dynamic execution of code that is written in PHP or Hack. The code is first transpiled into HHBC, a high-level bytecode format that serves as an intermediate language. This bytecode is dynamically executed by the HHVM. As a bytecode virtual machine, HHVM has similarities to V8, the JVM, or the CLR.

Keith Adams was an engineer at Facebook for six years, where he helped develop infrastructure to scale PHP effectively. Keith is now the chief architect at Slack, which is also a scaled PHP application. Keith returns to Software Engineering Daily to discuss why and how Facebook scaled PHP.

We have a new app for iOS and Android. This app is a great listening experience for Software Engineering Daily. It includes all of our old episodes, as well as comments and social features. I'll be commenting on each of the episodes going forward for an extend period of time. So, if you hear an episode that you have some commentary on, you can jump on to the app or on to softwaredaily.com/which is our web platform, and share your thoughts. Have a discussion on what you liked or disliked about the episode or what stood out to you. If you want to become a paid subscriber, you can get ad-free episodes. You can go to softwareengineeringdaily.com/subscribe.

FindCollabs is the company I'm working on. FindCollabs is a place to find collaborators and build projects. FindCollabs is having an online hackathon with $2,500 in prizes. So if you're working on a project, or you're looking for other programmers to build a project, or collaborate with or start a company with, or make music with, you can check out FindCollabs.

With that, let's get on to today's show.

[SPONSOR MESSAGE]

[00:04:31] JM: When I'm building a new product, G2i is the company that I call on to help me find a developer who can build the first version of my product. G2i is a hiring platform run by engineers that matches you with React, React Native, GraphQL and mobile engineers who you can trust. Whether you are a new company building your first product, like me, or an established company that wants additional engineering help, G2i has the talent that you need to accomplish your goals.

Go to softwareengineeringdaily.com/g2i to learn more about what G2i has to offer. We've also done several shows with the people who run G2i, Gabe Greenberg, and the rest of his team. These are engineers who know about the React ecosystem, about the mobile ecosystem, about GraphQL, React Native. They know their stuff and they run a great organization.

In my personal experience, G2i has linked me up with experienced engineers that can fit my budget, and the G2i staff are friendly and easy to work with. They know how product development works. They can help you find the perfect engineer for your stack, and you can go to softwareengineeringdaily.com/g2i to learn more about G2i.

Thank you to G2i for being a great supporter of Software Engineering Daily both as listeners and also as people who have contributed code that have helped me out in my projects. So if you want to get some additional help for your engineering projects, go to softwareengineeringdaily.com/g2i.

[INTERVIEW]

[00:06:22] JM: Keith Adams, you are a chief architecture at Slack and you are a long-time Facebook engineer before that. Welcome to Software Engineering Daily.

[00:06:31] KA: Thanks so much, Jeff. Good to be here again.

[00:06:33] JM: You had worked at VMware for 8 years before you joined Facebook, and my sense is that the engineering problems and culture at VMware were significantly different than that of Facebook. How did VMware contrast with Facebook?

[00:06:50] KA: Wow! That's a really tough one. So, of course, the VMware that I experienced from 2000 to 2008 wasn't the same company the whole time I was there. Either I was there something like the 20th of 22nd engineer at VMware. By the time I left 8 years, there were many thousands of people with all kinds of roles.

Broadly, there was a bunch of differences in VMware both in terms of the kinds of technology it was building and the kind of business it was in with Facebook. So, in terms of the technology that VMware was building, VMware – I imagine your listeners are broadly familiar with VMware, but it's morphed a lot over the decades.

Back then, VMware was very purely focused on delivering virtual machines, delivering hardware level virtual machines that let you slice up a chunk of PC hardware into lots of little software pieces. Believe it or not, this was a big, exciting, technical thing to be able to do in the late 90s and early auths. There were published papers that said you couldn't do it for instance. There're lot of sort of high-flying stunt systems programming that made it possible that I found really exciting, because I was a really enthusiastic practitioner systems back then. Still am in some ways, but other frontiers moved on a bit.

The business that VMware was in was essentially selling hardware. It was hardware that was implemented in software, but it was software that people were going to put to the same uses that they put hardware to. That meant that there was a really high-bar for performance, for quality, for predictability, that are there's a ton of attention paid to release engineering and the kind of art of putting bits in customers hand that was close to flawless as we could imagine.

This also sort of before the days of delivering software with the internet and before continuous delivery and continuous integration. When we talked about the channel in software, then we meant a retail channel where shrink wrapped boxes of CDs would show up. There weren't DVDs yet. There were CDs.

So, in many ways, a different world than the world of delivering software over the internet that we sort of assumed by default that today. That focus on, first of all, being at the hardware level abstraction-wise, but also being at the hardware level in terms of kind of reliability and predictability of the things we're giving to people was very, very different from providing a service over the internet in the environment of kind of open-ended consumer product, where you're doing something different every week than you were the week before where there are more users every week than the week before where the pattern that they're using it with is different in this place from that place just because of what happens we've caught on in Turkey versus what's caught on in New Zealand or whatever. Was a radical change for me and a really powerful education.

I think I apologized a lot to sort of old-time Facebookers for my first year at Facebook, because I, like lots of other somewhat seasoned software engineers who came to Facebook at the time, was really laboring under the delusion that I had a lot to teach these folks. Was really laboring under the delusion that I knew how to do things right and that there's this computer science out there that has first principles in it, and that I knew how to kind of write good code and test and all of these stuff in ways that they didn't.

The fact that sort of a lot of the things that were going on at Facebook seemed crazy to me at first initially sparked a lot of curiosity, but also a lot of skepticism on my part. It seems like from what I knew, none of these could work, and yet there it was. I could touch it in a web browser. I could load Facebook over and over again and it kept working. Even though almost everything about the software practices was sort of upside down and backwards from what I had experienced so far.

In terms of the things that just seemed radically different at the time. At the time I left VMware, I think the code I was writing the week I left VMware probably got into customers hand two years, maybe a little bit more after I left VMware. That was actually the cycle of learning. That was how long it took you to find out whether your ideas were good or bad. That was how long it took you to find out whether the thing that you thought was going to be great was actually so great in practice.

By the time those two years have passed and a friend of mine who's still at VMware emailed me saying, "Hey, we finally did that release." I've been through so many iterations and tried so many different things at Facebook that I was just like, "Oh! I see. The sort of cycle of being able to try things more times in a career is so powerful." Even if kind of the raw economy wasn't drawing a lot of engineers over to the world of sort of consumer services at the time, just the sheer pleasure and pace of being able to learn something in a couple of weeks instead of a couple of years was so wonderful that I'd have happily done that forever form then on.

[00:11:32] JM: What was your first project at Facebook when you joined the company?

[00:11:36] KA: Yeah. So when I first started – Before I started, first of all, I had talked to a lot of people on the Memcache team at Facebook. I was reading a lot of Memcache source code and stuff before I got started at Facebook and assumed that was what I was going to work on. But then when I got to the place, so Facebook already had instituted its sort of famous boot camp system by then. I think at the time it was presented to me as, "Hey, here is how we do engineering hiring. All engineers start out in this boot camp group. The boot camp team is a team. It's just a team that has high churn and you fix bugs for a while until you figure out the lay of the land and figure out which teams can use you and which teams are appealing to you as well."

So it was kind of this stable marriage between new boot campers and teams that need personnel. This was presented to me as just sort of the way things were done. I later found out that I was the second boot camp class. I later found out that this was all kind of somewhat theatrical that they're presenting this as like a big institution but had been working for a long time.

But the institution had its intended effect with me. I actually ended up somewhere different than I expected. So a couple of weeks in I met Aditya Agarwal, who's better known for lots of other things now probably. Most recently was CTO of Dropbox. But Aditya ended up being my first boss at Facebook and he was running the search team at the time. The search backend was a little bit long in the tooth and there was also some product plans around this time.

So, to give you some idea, this is early 2009, and a kind of big exciting feature on the web at the time was like auto complete in search boxes. That felt kind of new. The idea that you type a few keys, it would actually be doing a little AJAX request in the background to kind of fill in what you're typing keys. Saying AJAX, I'm dating myself here.

We want to – Zuck was excited about this. We wanted to do that, but for people search on Facebook. The actual sheer numbers involved were pretty stimulating, right? This was a long time ago. So it's a lot fewer users. I think we were celebrating round 200 billion monthly active users at the time. So, kind of more than a factor of 10 off of what there are now.

Still, 200 million people over small numbers of hundreds of milliseconds across the whole wide world seemed pretty challenging and interesting. So I ended up spending the first couple of years at Facebook working on the search backend and, specifically, the parts of the search backend that supported this system.

Kind of a side here, I can give you a link to a TED Talk I gave around then after we shipped that system. That was a great experience with this kind of quick iteration. We tried lots of different data structures. Lots of different strategies for how we shard the data and things like over the course of just 6 or 8 months.

[SPONSOR MESSAGE]

[00:14:16] JM: Today's show is sponsored by Datadog, a monitoring and analytics platform that integrates with more than 250 technologies, including AWS, Kubernetes and Lambda. Datadog unites metrics, traces and logs in one platform so that you can get full visibility into your infrastructure and your applications. Check out new features like trace search and analytics for rapid insights into high-cardinality data. Watchdog, an auto-detection engine that alerts you to performance anomalies across your applications. Datadog makes it easy for teams to monitor every layer of their stack in one place.

But don't take our word for it. You can start a free trial today and Datadog will send you a t-shirt for free at softwareengineeringdaily.com/datadog. To get that t-shirt and your free Datadog trial, go to softwareengineeringdaily.com/datadog.

[INTERVIEW CONTINUED]

[00:15:22] JM: The data structures in the search application for Facebook, I can see why that would be a non-obvious set of problems to tackle, because Facebook data can be modeled in so many different ways. You can model it as a graph, or an inverted index, or an LRM cache. What data structures did you find to be most useful?

[00:15:50] KA: Yeah. For this backend, the right answer was a cache – Or excuse me. Was a graph for us. So, the most powerful feature that we had in our hands for ranking results was just distance and the social graph at the time. So, I remember, I used to – At least when there were 200 million users, there are about 700 people named Keith Adams on Facebook. It's a very common last name. Pretty common first name. So lots and lots of Keith Adams are out there.

Everybody at work who typed Keith Adams that typed ahead, I really, really wanted me to be number one with the bullet, because there's enough information out there that I'm the one they're talking about, because we lived in Menlo Park and because we work in Facebook and because we're in some of the same groups and so on and so forth. But most especially because we're not that far apart in the social graph.

The way that we kind of structured things, we didn't actually bother representing the user doing the query beyond just sending along their first degree connections. The frontend web application would have your direct connections in order to do anything. One of the first it did for every user interaction was just go out to Memcache and fetch your friend list.

So the query would actually throw my user I.D. away. My user I.D. doesn't help at all. I know who I am and it's not going to help with the search. But what does help with the search is who my friends are. So if you did a little prefix query that was like JEF and some other person on the other side of the world happened to have all the same friends as you, they would actually get the same results as you. So the query was basically some tiny snippet of text, but most of the information of the query was actually your first degree edges. This is all kind of hidden in the backend. It wasn't sort of – That stuff wasn't going out over the internet.

[00:17:24] JM: Eventually, you made it into a situation where you're working on PHP infrastructure, and this was some of your most influential work at Facebook. How did you find yourself working on PHP infrastructure?

[00:17:38] KA: Yeah. So, for those that don't know, Facebook was and is most respects a kind of overgrown LAMP stack app. So around the time I started, it was still recognizably your father's or your mother's LAMP stack application scaled out quite a bit. You had Linux Apache, MySQL and PHP. The MySQL part of it, obviously we weren't on a single MySQL database anymore. We were doing client routing and client sharding in PHP to figure out which database to get a datum from or to update a datum in.

I think at the time, we're on the order of small numbers of millions of lines of PHP. So, good size program. Lots of complicated things can happen in small numbers of millions of lines. But the thing that was really getting our attention was that we were burning a ton of CPU executing this PHP, and this had two problems. The first problem was just that PHP is organically a single-threated language. There's a little bit of nuance on that. But it's close to true. That's just a single fettered language.

If you are burning lots of CPU, that means you're also going pretty slow. A single threat can only consume so many CPU cycles. So the site felt sluggish. It was feeling more and more sluggish, and there was a multi-pronged attack on that happening.

There's another piece of this, which was actually economic. So, at the time, this is kind of hard to imagine, but even as late as 2009, it wasn't manifestly clear to everybody what Facebook's business was going to end up being. There were some experimentation with like gifts. Maybe gift giving would be an organic activity. People noticed that people liked wishing each other happy birthday on the site for instance. The thought maybe giving people birthday gifts and taking a cut of that would be the thing to do. Virtual gaming - Gaming was starting to happen and virtual currencies in gaming were starting to happen. So maybe that was going to be really important.

Around the time of the IPOs, it was something like 30% of Facebook's revenue. So that did end up being important. But it wasn't clear yet just how expensive it would be to run the site for a

given user that we could possibly break even. So, as far as 2009, which is a good many years into Facebook's history, from an economic point of view, Facebook was just this big hole in the ground that VC dollars had been ported to. It was just sort of this thing that seemed to get a lot of usage, that seemed very popular, attracted a lot of talented people. Everybody felt like there might be something there. But it wasn't obvious where you'd get the money from.

A big component to where those VC dollars were going was CPUs. We needed to do something to reduce the CPU cost of running the site on PHP. If you've never had an experience sort of trying to do heavy compute in PHP, that is sort of not the express purpose of PHP. If you're doing heavy literary algebra or graphics or whatever. That is not a domain that PHP specializes in deeply.

If you would complain about this problem publicly at the time, this is back when Facebook's engineering brand was a lot weaker than it is now. If you kind of talked about any of these problems on Hacker News or somewhere, you'd usually get sort of some internet guy on Hacker News saying something like, "Well, yeah. You're using PHP wrong. It's a glue language for connecting database queries. It's supposed to actually compute in it."

But as Facebook was finding and lots of other places have found since then. I mean, this is true at Wikipedia. It's true for Zynga, for lots of other kind of heavy users of server side PHP were actually had to scale out the backend. You end up having to do joins in a language. You end up having to do sort of some database joins, where the join condition is essentially a Turing complete. It's some application level of construction. The only thing that knows what in Facebook what a user is and who their friends are and what a photo is and what the permissions on the album are and these other sorts of natural join conditions is the language itself. So you end up sort of fetching these big lists in PHP and then intersecting them with this sort of complicated logic and having only a handful of things come out at the intersection.

This was viewed as sort of a pattern that kept recurring. It wasn't going anywhere. It was core to the value that Facebook provided was that it connected all these kind of different people and types of media together. We're going to take lots of – Try lots of different things in parallel to try to get our hands around that.

One of the things that people were trying in parallel was to make PHP faster. So, there is a really important – The project I ended up working on to do that was called HHVM. But to sort of explain the context that HHVM rose in, there was a really important predecessor effort that HHVM built on. This is called HipHop, or within Facebook, HPHP.

HPHP was a project that was started initially by Haiping Zhao, but significant team grew up around Haiping to do this. HPHP was sort of what people call a transpiler these days. It took a PHP source base's input and tried to produce a semantically equivalent C++ program. This might sound like a strange thing to do, but observe that – Well, I mean, maybe it doesn't sound like a strange thing to do.

PHP is slow. C++ is fast. If you can lower PHP down to C++, maybe you get a fast program out of it. That's not an inherently crazy thing to attempt. The project – To my continued amazement. I still find it really heroic that it worked as well as it did it. They built a thing that actually worked, that could take a big complicated PHP program like Facebook at the time. Compile it to this completely different domain with a very different set of expectations, very different runtime environment. Compile it, link it, produce a gigantic binary, and that binary [inaudible 00:23:20] port 80 of Facebook for several years and bought Facebook a lot of runway.

[00:23:28] JM: No. No. I mean, we could just pause and just reflect on how unenviable a proposition that would seem if somebody came into my office and said, "I've got a great idea for your next project. You're going to transpile every PHP application into C++. Go." You know, it is laudable that a team was actually able to do that and it provided meaningful business value.

[00:23:53] KA: Oh, it absolutely did. Out of the gate, it was order of 2x faster. Therefore, 2x more economical in terms of dollars per user spent on CPUs. That bought us, the company, Facebook, years of runway.

[00:24:08] JM: So just for people who are not super familiar with this idea, and I'm sure that's a lot of the audience. What were some of the shortcomings of that approach? When you were looking at that approach, what opportunities for improvement did you see?

[00:24:23] KA: Right. So, one of the things that's an issue with that is if the transpiler version of HPHP were the only thing that existed, you'd have a very difficult development cycle, because one of the beautiful things about PHP is that it's very sparing on developer productivity. It's very respectful of the developer time.

So the normal PHP workload is that you hit save in your editor and you reload the page. There's not even like a web application server to restart, because the kind of application model of PHP is that each web request starts from nothing.

PHP was kind of AWS Lambda before AMS Lambda came along. It has an execution model that's very similar to the draw backs and upsides of Lambda. So, you go from that to a world where my workflow is now save. Invoke the HPHP compiler to translate this large source base into an equally large, if not larger C++ source base. Invoke the entire C++ tool chain on that source base including a linker that produces a multi-gigabyte binary.

Fun story at Facebook for a while in the late auths was driving some bug fixing in the GNU linker tool chain, because we were hitting limits the limits [inaudible 00:25:34] to represent binary sizes, and we seemed to be the first people to hit. The way we hit it was by turning our multi-million line PHP program into a more multi-million line C++ program and then trying to link the whole thing into one big stack binary. This was unacceptably slow, but the compile times were in the order of a half hour at that time. It meant that you needed to have some other runtime environment for just playing around it. You needed to have some kind of iteration cycle that was a lot shorter than that.

So you still had a PHP interpreter, but not the PHP interpreter works really differently from the way that the actual production environment works. So you have a set of difficulties of kind of making sure that you have a set of prods that appeared where the dev environment is sort of so different from prod that bugs appeared one of the other and don't appear on the other side, and that led to some really hard debugging situations.

But more to the point, this is kind of where it starts to dovetail with my background at VMware. There's a history of trying to run heavily dynamic languages, like PHP efficiently on hardware.

That history nudges us strongly in the direction of using runtime feedback to run the language efficiently.

So while this sort of land speed record holding languages today are all statically compiled, like Rust and C++, all the efforts we have to run languages like JavaScript or Python or, now, PHP, efficiently seem to need runtime feedback. They're all JITs, right? They don't produce a static binary.

The reason that's true is that there are limits to what you can analyze ahead of time for these languages. You can't bound the set of values that are going to actually flow through the data flow and control flow graph of this thing as well as you can with a C++ program. So, your C++ compiler ahead of time can make decisions about what goes in a register. What goes in a floating point register? What the target of a branch is going to be? What the target of the call is going to be in ways that just are impossible for very simple computer science reasons for a PHP for a JavaScript program.

When I was at VMware, the team that I was in was led by a man named Ole Agesen, Ole Agesen is actually still at VMware by the way. He's a distinguished engineer there and just is one of the most remarkable engineers I've ever had the pleasure of working on. Ole had come from a background of working on self and small talk. There've been a community before sort of object orientation was cool that was trying to run these languages like self and small talk that were very dynamic and very aggressively object oriented in an efficient way.

They sort of were that little nucleus of people from Arhus Denmark, including Ole Agesen, was a kind of a formative place for the future of just in time compilation. So, Ole ended up at VMware, because VMware's approach to doing hardware level virtualization on the x86 actually involved JIT compilation.

All right. So, a colleague of his form then was Lars Bach who ended up pioneering V8 for JavaScript and both of them had DNA from the self and small talk communities and had worked on some of the same projects.

So, I'd been a little bit steeped in this – At the time, somewhat obscure world of people trying to run dynamic languages fast at my previous gig. I had a sort of athletic respect for what HPHP had achieved, who's a virtuoso feat of engineering and C++ programming. But I also had a conviction that there was something missing from that approach that we'd need runtime feedback.

I started talking about this conviction of mine with a couple of other folks at Facebook at the time. One of them was Jason Evans. Jason Evans is an extraordinary systems programmer. Facebook came to know of him through JE Malloc, which is a Malloc Free implementation that he wrote that is wonderful for scalable over memory efficient, returns memory to the operating system in a great way. Was working better for Facebook at the time in production than TC Malloc, which is an allocator with similar goals was.

We hired Jason Evans, and I just like talking systems programming with him immediately. So I kind of unspooled this story for him about like, "Wow! HPHP is amazing, but there's got to be something it's missing because of all the stuff that it can't resolve ahead of time."

We ended up propping into these conversations, a fellow named Drew Peroski, and Drew Peroski had come to us from Microsoft, where he'd work in Microsoft's dev on tools related to the CLR, which is common language runtime, Microsoft's runtime for .NET and associated ecosystem. All the languages that run on the .NET platform.

The three of us started talking more and more about this and it started to come together into something that we'd released HPHP as an open source project. We had christened it HipHop for public consumption, because of trademark reasons. PHP is actually a registered trademark. So if you go around calling your project something-something PHP. You'll get a sternly worded letter.

So if you grabbed /usr/dict/words for words that start with H and then have a P and then an H and then a P, HipHop is by far the most palatable one you'll come up with. So, HipHop had been sort of released as an actual project, and me, Jason and Drew started talking about something that eventually before it had an real identity, we started calling the HipHop VM.

To kind of convince myself that there was something here, I built – During hackathon actually in, I want to say, late 2009, maybe early 2010, I've built a hackathon project that transpiled a tiny subset of PHP into JavaScript and run it under V8, which had been released already.

In retrospect, that experiment, that kind of experiment is not very convincing to people who don't already want to believe you. You're running micro-benchmarks. There's all kinds of ways to cheat. In retrospect, there were all kind of hard things about PHP that my little prototype just completely punted. But it felt like something we could reach out and touch and felt like something that you could see running PHP faster than we were running PHP, and that felt like a sort of sweet motivational taste.

The approach we ended up taking had nothing to do with that, and the hackathon project as a code artifact was thrown away. But as a sort of interactive thing that you could feel and get excited about I think was really useful for me and Drew and Jason at first.

[00:31:52] JM: Just to make sure I understand this correctly. So, HHVM, in terms of the prototype that you made, was a way of running PHP on the V8 JavaScript engine.

[00:32:08] KA: Yeah. I don't think we're calling it HHVM at that time. I think there were two things going on. One was me, Drew and Jason started saying, "A JIT could do well with PHP for first principles reasons, and wouldn't it be cool if there was this system that was like HipHop, but was the HipHop virtual machine instead of being a fully ahead of time system that way HipHop as it was then was."

The other thing that was going on was Hackathon came up and I wanted to play around with convincing myself there was something here. So I don't know if that – It's a night's work. It was a throw away effort. I don't think I ever bothered naming it. But that little effort, even though it's not real science and isn't sort of publishable and doesn't tell you anything about the language PHP and how that would really run if you tried to repurpose V8 to run PHP. Convinced us that there was something there, and you could see something that at least looked like PHP running a lot faster than we were able to run PHP.

[00:33:02] JM: Right. We've done some shows about JavaScript engines, and you do see some emphasis on the fact that it's a just in time compilation environment rather than this entirely ahead of time compilation world, like C++. Intuitively, it makes sense to me – If I understand correctly, it's the same thinking that you were coming from. Intuitively, it makes sense to me that there will be some symmetry between what you want to optimize for with the PHP, getting PHP fast versus – On JavaScript, versus – Like that would be more symmetrical getting PHP to run quickly in a JavaScript VM versus running quickly in a C++ environment since that's an ahead of time compilation environment.

[00:33:53] KA: Yeah, I think you're exactly right, and there's – My thinking about this has changed a lot over the years in part because of the experience that I had for about four years working on HHVM. But the core problem of trying to run a language like JavaScript or PHP or small talk or self-fast is just the level of dynamism in the runtime. Just that the binding of names to sort of code locations is so fluid and can differ from run-to-run. In a very real sense, JavaScript programs and PHP programs kind of link themselves at runtime every time they run. They kind of pull in the parts that they're going to run and then bind names to those parts and start executing them. To be clear, there are –

[00:34:34] JM: V8 optimizes for that fact.

[00:34:37] KA: Exactly. Yeah. So a runtime that expects that is going to have tricks up its sleeve that a C++ runtime just doesn't have, because it has no reason for them.

[00:34:46] JM: Yeah. So, how did that kernel of knowledge from the hackathon prototype, how did that turn into something that is not as mature as HHVM?

[00:35:00] KA: Yeah, that's a great question. So at the time, it was actually controversial sort of what we should do and how we should do it. I think as Drew and Jason and I got more excited about the idea, the core idea, it felt like we had a couple of ways to pursue it. One way to pursue it was to try to repurpose V8 in some ways. So, give V8 a PHP frontend or give it the ability to run PHP more deeply.

That would have been a mistake in ways that we actually weren't fully equipped to appreciate at the time, but we can touch on later if you want. There's a great paper called the repurposed JIT phenomenon, about the fact that it's a lot of hard work to write a JIT. So, when people come up with a new language that they wished they had a JIT for, they try to find some way to get another JIT to do the heavy lifting for them.

This usually happens with the JVM or the .NET CLR. For instance, JRuby is an example of a JIT for Ruby, where they try to repurpose the Java Virtual Machine to do a good job for them. JRuby is a wonderful project. Whenever I talk about JRuby as an example of the repurposed JIT phenomenon, Charles Nutter, who's JRuby's benevolent dictator, shows up and accuses me of all kinds of intellectual dishonesty. Bit asterisk here. Go read everything Charles Nutter has to say about JRuby if you're skeptical.

But I think there are statistical realities about Java programs that the JVM optimizes for. The JVM is there to run Java, and to some extent, Scala programs well. The design of Scala was influenced by the design of the JVM. Scala is the language that it is, and instead of just being F# for instance. In part because it has to run well on the JVM, right?

There were a million things about PH that would just have been terrible fits for V8 that I wasn't even aware of at the time. Probably the biggest one is that the runtime for PHP is reference counted, whereas V8 was optimized around tracing garbage collectors. Just to dive in here for those that aren't necessarily automatic memory management buffs. There are two kind of big approaches to automatically reclaiming memory from a running program without explicit for ease in it, right?

One approach is trace the graph of objects that can be reached. Anything that hasn't been found by that tracing, you know it's garbage, because there's no path to it. We claim all that memory. That raw approach is associated with controlling pause times. If you are interested in doing a good job with the tracing GC, your core problem is controlling pause time. There are good ways to control pause time, but that's the core problem.

The alternate approach is reference counting. In reference counting, each object maintains account of how many inbound errors it has. Keep those counts up to date as the graph of

references into a program change. Whenever goes to zero, you know that it's garbage and you can reclaim it.

The core problem with reference counting is keeping the cost of updating those counts controllable, because almost every time that you touch anything, you're running around writing chunks of memory that you otherwise wouldn't really have to be writing.

So, if you were to try – The crazy thing about PHP is that it exposes the semantics of reference counting to PHP programs. So if you try to make a PHP runtime on top of the Java Virtual Machine, or V8, or the CLR, or pick your favorite JIT runtime here, you're probably going to be running reference counting with all of its problems on top of tracing collection with all of its problems. You'll have a kind of multiplicative effect of the overheads that you face usually. This has been the reason that sort of repurposed JIT has burned PHP especially hard historically. So, it's good we didn't try to reuse V8.

One option we had was to try to go our own way entirely. So, start up a new project. I knew runtime for PHP maybe be based on the sort of mainstream open source PHP engine, maybe not. The feeling then, which I think in retrospect was correct, was that that was an awful lot of unrewarded work to take on in the Facebook environment, where HPHP was successfully running the whole complicated application that was our actual target.

So, there had been a ton of extensions rewritten into HPHP. It was beginning to export some language extensions. So it brought yields, Python styles generators. As far as I know, first appeared of any PHP variant inside of Facebook as part of HipHop, and it felt like it was going to be important to track HipHop really closely in part just because that was going to be what the application we cared about was going to be running on.

So we made the decision to build the HipHop VM as a physical extension within the same source tree as HipHop the ahead of time compiler. So there'd be this one way you could invoke HPHP where it would be an ahead of time compiler. There's this other mode you could run it in where it was a VM with a full runtime and full compilation stack inside of it.

[00:40:00] JM: Wait. What was the advantage of having those two different paths that it could take?

[00:40:06] KA: The advantage there was just the amount of sort of extensions to the language, so to the frontend of the compiler, parsing and so forth and pieces of the runtime that we could use. So, HipHop had a working PHP reference counting memory management system that we wanted to piggyback on. But it also had a bunch of life support in the form of a web server that was integrated. So PHP is usually used as part of a web server. HipHop also replaced Apache in LAMP Stack. So it was really kind of the LHM Stack instead of taking out both the A and the B in LAMP Stack and sort of server configuration would have been the same. So, there was a lot of nontrivial stuff that HipHop already had working that seemed like we had nothing special to add to.

[SPONSOR MESSAGE]

[00:40:57] JM: When you start a business, you don't have much revenue. There isn't much accounting to manage, but as your business grows, your number of customers grows. It becomes harder to track your numbers. If you don't know your numbers, you don't know your business.

NetSuite is a cloud business system that saves you time and gets you organized. As your business grows, you need to start doing invoicing, and accounting, and customer relationship management. NetSuite is a complete business management software platform that handles sales, financing, and accounting, and orders, and HR. NetSuite gives you visibility into your business, helping you to control and grow your business.

NetSuite is offering a free guide, 7-key strategies to grow your profits at netsuite.com/sedaily. That's netsuite.com/sedaily. You can get a free guide on the 7-key strategies to grow your profits.

As your business grows, it can feel overwhelming. I know this from this experience. You have too many systems to manage. You've got spreadsheets, and accounting documents, and invoices, and many other things. That's why NetSuite brings these different business systems

together. To learn how to get organized and get your free guide to 7-key strategies to grow your profits, go to netsuite.com/sedaily. That's NetSuite, N-E-T-S-U-I-T-E.com/sedaily.

[INTERVIEW CONTINUED]

[00:42:49] JM: Let's take this as an example of what Facebook was doing differently than some of the other companies in Facebook's vintage. So, the easiest status quo to draw from that time is Google. The status quo for a hyper-scale company was Google, and Facebook did things differently than Google in terms of scalability both for technical reasons, but also for maybe cultural reasons or just – I don't know, deliberate reasons, product reasons.

How did Facebook's strategy and Facebook's engineering culture differ from that of Google or from whatever else you would define as the status quo at the time?

[00:43:35] KA: Yeah, it's a really interesting question. I should preface this by saying I've never worked at Google. It seems like there are a lot of brilliant people there and there were a lot of brilliant people there who've done a lot of great work. I don't think we necessarily got something right that they didn't understand or anything. But it's true that there was a different house style in various ways, and I think one of the ones that probably jumps out at me is just this – The whole reason that the HHVM project felt like it had a reasonable motivation was the existence of this large and growing PHP monolith.

I think in the house style of Google at the time, while they weren't calling them microservices, and that's sort of vocabulary around microservices that we use today would have been a little alien at the time. I think from the outside, my impression is that you could have thought of Google as a sort of C++ microservices shop around that time.

So, if they found themselves sitting on a 4 million line application that was sort of tightly integrated and was just calling functions all over the places and passing data all over the place and sharing one big complicated set of reasoning about why it all works in one big suite of tests and so on. I think that would have esthetically odious to a lot of people inside of Google. To be clear, all else being equal, it is a little esthetically odious, right? If there's no reason for it to be that way, it probably should not be that way.

So, I think relatively early on, Google had some success with heavily verticalized products, first of all. When Google Maps and Gmail and Google Web Search were sort of the only three things that people thought of initially with Google, those were three like completely different products. There was no log-in that you shared across them. There's no kind of layer of identity or data sharing that unified them. The UIs completely different.

So, a really sort of siloed development process where different teams had their own codebases, had their own release cycles, have their own tooling, shared some infrastructure maybe below the user visible level. But were the things listening on the other side of port 80 were very different depending on which product you were using, reflected the challenges that Google had and reflected the experiences they were trying to make happen.

The extent that there was something different about Facebook as an experience at the time, it's unity. It was the fact that Facebook was very identity-centered. It didn't make sense to do anything on Facebook not logged in practically. The abstractions and concepts that occurred in different Facebook products had some sort of coherence and unity to them.

So, the set of things that you could do with the status update, like like it and comment on it and share it and so forth were also the set of things that you could do with a photo, and were also the set of things that you could with the web link that you found and so on. There was a kind of embracing of that complexity in form of one large codebase that made it easier to make certain product experiences happen.

I talk a lot with people – This is still an important kind of concept, I think, that we live in a time where we're almost in a more is better microservices regime where people will tell you that sort of dissecting your product into independent services is a good in and of itself.

I think with all else being equal, that's probably true. But all else isn't always equal, and I use the product example a lot of times of Microsoft Excel, just because it's very universally accessible. Lots of people have used Excel. A surprising diversity of use cases in Excel as super well-supported. A lot of where that's come from has been – You think of sort of all the entire set of features in Excel. They've got N-features. They've got pivot tables, and they've got formulas,

and they've got visual basic, and they've got sheets, and they've got charts, and they've got the charts themselves [inaudible 00:47:34] thousands of features.

A lot of the value of Excel actually comes from just tackling the matrix of feature-to-feature interactions. Those N-squared, how does this work with that other thing? Microsoft actually paid PMs to sit there and think, "Okay. What should happen with the pivot table when there's a chart and I paste it into a thing?" They have gone through the hard first product labor and an engineering labor to think through how all that should actually work. Write the code to make sure it works that way and then put it in their monolith, then put it in their big ball of logic that actually determines what Excel is like.

Facebook at the time – And I think my current job with Slack's product as well has this characteristics, was more like that Excel experience than like the web search versus Gmail experience. I think there are companies and domains and applications where that's sort of heavily siloed, heavily separated think and work really well. But there are other applications that sort of want to be tightly integrated and organically grow to be so big that the tail does start to wag the dog, that you do start doing things like building your own web server or building your own programming language or both in the case of HHVM to better accommodate this very valuable, very hard to split up thing. I think that can be okay, and I'm not sure that we're – As an industry, as okay with accepting that as we ought to be.

[00:48:58] JM: There are a number of strategic inflection points in Facebook's history that the company was able to overcome through great engineering, through great product organization. So whether we talk about the shift to mobile, or the overcoming, the lack of a business model right after the IPO, or kind of contesting Google+, those kinds of different strategic inflection points.

There was something about Facebook that has made it very resilient to strengths, or very resilient to threats. I don't know, maybe it's the charisma of Zuckerberg, or just the energy that was in the company at the time for whatever reason. Do you have any sense of what is about the Facebook culture that has allowed it to come out on the other end of strategic inflection points when there are other companies that, when they hit a strategic inflection point, they can't overcome it and they perish.

[00:50:03] KA: It's an interesting question. I think I'm experiencing – Let's see. I want to answer this humbly, because my experience of Facebook was as an engineering, and while there were technical elements to how Facebook sort of executed its plan to get through those problems. Like the phenomena you're pointing out were I think more at the scale of kind of company leadership and strategy than anything I have privileged inside into other than the fact I happen to be there at the time.

I will say that sort of the first one of those that I felt like I weathered as part of the team at Facebook was Google+. So, circa 2009, 2010 or so, it was really clear that Google was – That a drumbeat was building inside of Google to destroy Facebook, and it wasn't sort of to get into social networking, or destroy Facebook and Twitter and Bebo and MySpace. It was clearly focused on us.

There were a lot of kind of varying levels of white and gray hat, things that were happening in the interactions between those companies. One of them was that there was a lot of kind of aggressive scrapping in the social graph going on on the part of Google seemingly feeding back into their products that they're building.

When credible word about sort of what would eventually launched as Google+ kind of reached us. I remember it feeling – When I first heard that news, I remember it feeling really scary, and this is kind of a hard thing to cast your mind back to 10 years now. But it wasn't clear Facebook was going to win or whatever. Google was enormously larger, enormously more resourced, seemingly willing to do anything to defeat us and has many engineers that has worked at Facebook at the time hold up building this straight up Facebook clone.

So, they'd had a couple of sort of misfires already. They had Google Buzz, and to some extent, Google Wave, although that was a sort of orthogonal attack in some ways. With Google+, what was sort of alarming for us was like there's no pride or sort of – There's no claim that they like had some insight into how to do it right when Facebook was doing it poorly. It just was a Facebook clone. It just was a rip-off. Except they were going to make you log into it to search the web. That was very terrifying when we first heard about it.

The thing that I want to give a lot of credit to for Facebook leadership with our reaction to Google+ at the time was the balance between reaction and overreaction that they managed to strike. There is new information in the fact that a competitor is doing this, and it would be crazy to ignore it. It unleashes this kind of nervous energy in the workforce too, right? We're all very bought in. We all want our Facebook stock to be worth something someday. So, there's a kind of potential energy there that you'd like to harness and that you'd like to use, but you still have a strategy that you're trying to execute on, and you still have a roadmap that you need to get down and you still need to grow and you still need to keep the lights on.

The balance between kind of messaging this in a way that accepted the challenge, but framed it as an adventure and a journey we were on together and a challenge that we were going to rise to as supposed to just the end of the world, that there's this gorilla that's enormously more powerful. All the engineers who work at Google are 10-feet tall and eat razorblades for breakfast and they're all going to come destroy us. That struck as a really important component of it was just the internal messaging of this is – If we make through this, this is going to be the greatest story that we're ever going to be able to tell about our careers.

I hope I have a few better stories now, but it sure was the topper for several years. Yeah, I'm not sure that you'd want to kind of paint a mosaic of this. I'm sure that other people would have different answers to this. A computer programmer, some hot shot programmer, is not the person who probably has all the insights into this. But in retrospect, that was really an impressive display of leadership on Facebook's part at the time.

[00:54:07] JM: All right. Well, it will be one part of that mosaic. Last question, or a couple of questions. The constraints around Facebook when you joined in 2009, those are very different operating constraints than building a company today, in 2019. Now that you're at Slack, you have access to much more technology that we had 10 years ago. What are the most notable ways in which scaling a software company has changed?

[00:54:36] KA: That's hand down public cloud by 100,000 miles. The amount of energy and sort of long-time leads that went into buying a real estate and pouring cement on it and running power out to it and building it out versus what the public cloud offers is an entirely different world.

So, getting to – So Slack has been public cloud native for its entire life. In principle, Slack could someday scale to the point where we're getting off of public cloud, running in its own data centers could become economically viable. It's not something we eliminate. We want our customers to view us as a service and it's sort of the location that that service is physically housed in to be an implementation detail that they don't have to care about.

But it's been astonishing how far we've gotten without that even coming close to being a reasonable tradeoff of sort of convenience and flexibility versus the public cloud so far. I think there's a real generational shift between the era of startups where it was originally Mark Stoermer with a Linux server under his desk, and then it was a colo facility somewhere, and then it was a bigger colo facility, then we had to buy a data center, and so on. The capital intensiveness of that and the longtime leads of those kinds of scaling are just utterly a thing of the past now because of public cloud.

[00:55:53] JM: Any notable differences between the engineering culture at Slack versus that of Facebook?

[00:56:00] KA: Yeah. I think, in many ways, we started of talking about VMware, and then we talked about Facebook. In some ways, Slack is a business model that's closer to VMware. We're an enterprise software company combined with a tech stack that's closer to Facebook's. We're running in the cloud, running HHVM server side, running MySQL for storage and so on.

I think the engineering culture, in many ways, it comes out of – Your company culture is a complicated nonlinear function of the culture of your founders. The founders of Slack, it wasn't their first time at the rodeo. They've had a very successful startup in the past in the form of Flickr. They were at a different phase in their lives. That experience of having done something successful before reduces some of the paranoia that this is the only good idea you'll ever have.

There's a general feeling I think of – There's an attitude that since we are in the business of providing a service to customers who directly pay us, that surprising customers, or upsetting them, or doing something with their data that isn't exactly what they imagine we would do with their data is hugely unacceptable.

Not that those things were ever that acceptable on Facebook. There's this trope about Facebook that you're the product there, and it always really frustrated me when I was at Facebook. You're not the product of Facebook. You're the audience. It's like television. They're broadcast television. You don't pay a fee for. There are ads that support it, and that's where the money comes from. But it's not quite like you're the product of broadcast television.

If broadcast television can't manufacture more of you, they can't store you in a warehouse, they can't ship you from place-to-place. They can't decide to build you in some different way tomorrow. You really do need to choose to tune in for broadcast television to have any value and the same is true with Facebook.

But I think being in this business of directly taking your customer's money and then being completely free to leave tomorrow if they don't like you changes your relationship to customers in ways that I find great, honestly. I think this is a much simpler relationship to think about as sort of two-way business transaction as supposed to a three-way business transaction.

[00:58:07] JM: Keith Adams, thanks for coming back on. It's been really great talking to you.

[00:58:09] KA: Thanks, Jeff. Likewise.

[END OF INTERVIEW]

[00:58:15] JM: GoCD is a continuous delivery tool from ThoughtWorks. If you have heard about continuous delivery, but you don't know what it looks like in action, try the GoCD Test Drive at gocd.org/sedaily. GoCD's Test Drive will set up example pipelines for you to see how GoCD manages your continuous delivery workflows. Visualize your deployment pipelines and understand which tests are passing in which tests are failing. Continuous delivery helps you release your software faster and more reliably. Check out GoCD by going to gocd.org/sedaily and try out GoCD to get continuous delivery for your next project.

[END]